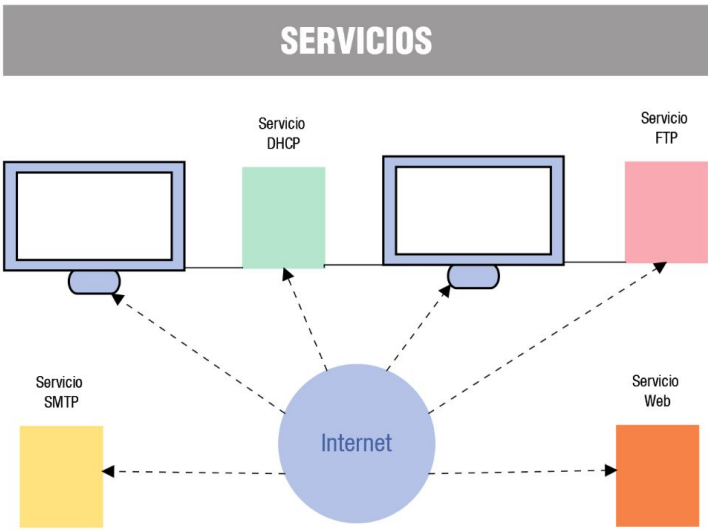
GENERACIÓN DE SERVICIOS EN RED

1.- Introducción.



Una red de ordenadores o red informática la podemos definir como un sistema de comunicaciones que conecta ordenadores y otros equipos informáticos entre sí, con la finalidad de compartir información y recursos.

Mediante la compartición de información y recursos en una red, los usuarios de la misma pueden hacer un mejor uso de ellos y, por tanto, mejorar el rendimiento global del sistema u organización.

Las **principales ventajas de utilizar una red de ordenadores** las podemos resumir en las siguientes:

- Reducción en el presupuesto para software y hardware.
- Posibilidad de organizar grupos de trabajo.
- Mejoras en la administración de los equipos y las aplicaciones.
- Mejoras en la integridad de los datos.
- Mayor seguridad para acceder a la información.
- Mayor facilidad de comunicación.

Pues bien, **los servicios en red son responsables directos de estas ventajas.**

Un servicio en red es un software o programa que proporciona una determinada funcionalidad o utilidad al sistema. Por lo general, estos programas están basados en un conjunto de protocolos y estándares

Una **clasificación de los servicios en red** atendiendo a su finalidad o propósito puede ser la siguiente:

- **Administración/Configuración**. Esta clase de servicios facilita la administración y gestión de las configuraciones de los distintos equipos de la red, por ejemplo: los servicios DHCP y DNS.
- **Acceso y control remoto**. Los servicios de acceso y control remoto, se encargan de permitir la conexión de usuarios a la red desde lugares remotos, verificando su identidad y controlando su acceso, por ejemplo Telnet y SSH.
- **De Ficheros**. Los servicios de ficheros consisten en ofrecer a la red grandes capacidades de almacenamiento para descongestionar o eliminar los discos de las estaciones de trabajo, permitiendo tanto el almacenamiento como la transferencia de ficheros, por ejemplo FTP.
- **Impresión**. Permite compartir de forma remota impresoras de alta calidad, capacidad y coste entre múltiples usuarios, reduciendo así el gasto.
- **Información**. Los servicios de información pueden servir ficheros en función de sus contenidos, como pueden ser los documentos de hipertexto, por ejemplo HTTP, o bien, pueden servir información para ser procesada por las aplicaciones, como es el caso de los servidores de bases de datos.
- **Comunicación**. Permiten la comunicación entre los usuarios a través de mensajes escritos, por ejemplo email o correo electrónico mediante el protocolo SMTP**.**

A veces, **un servicio toma como nombre, el nombre del protocolo del nivel de aplicación en el que está basado**. Por ejemplo, hablamos de servicio FTP, por basarse en el protocolo FTP.

En esta unidad, verás ejemplos de cómo programar en Java algunos de estos servicios, pero antes vamos a ver qué son los protocolos del nivel de aplicación.

2.- Protocolos de comunicaciones del nivel de aplicación.

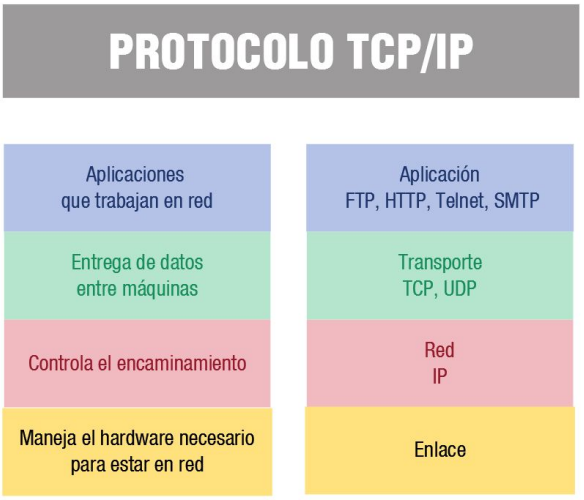
En la actualidad, el conjunto de protocolos TCP/IP constituyen el modelo más importante sobre el que se basa la comunicación de aplicaciones en red. Esto es debido, no sólo al espectacular desarrollo que ha tenido Internet, en los últimos años (recuerda que TCP/IP es el protocolo que utiliza Internet), sino porque también,

TCP/IP ha ido cobrando cada vez más protagonismo en las redes locales y corporativas.

Dentro de la jerarquía de protocolos TCP/IP **la capa de Aplicación ocupa el nivel superior y es precisamente la que incluye los protocolos de alto nivel relacionados con los servicios en red** que te hemos indicado antes.

La **capa de Aplicación define los protocolos (normas y reglas) que utilizan las aplicaciones para intercambiar datos**. En realidad, hay tantos protocolos de nivel de aplicación como aplicaciones distintas; y además, continuamente se desarrollan nuevas aplicaciones, por lo que el número de protocolos y servicios sigue creciendo.

En la siguiente imagen puedes ver un esquema de la familia de protocolos TCP/IP y su organización en capas o niveles.



Pila de protocolos TCP/IP.

A continuación, vamos a destacar, por su importancia y gran uso, **algunos de los protocolos estándar del nivel de aplicación**:

- **FTP**. Protocolo para la transferencia de ficheros.
- **Telnet**. Protocolo que permite acceder a máquinas remotas a través de una red. Permite manejar por completo la computadora remota mediante un intérprete de comandos.
- **SMTP**. Protocolo que permite transferir correo electrónico. Recurre al protocolo de oficina postal POP para almacenar mensajes en los servidores, en sus versiones POP2 (dependiente de SMTP para el envío de mensajes) y POP3 (independiente de SMTP).
- **HTTP**. Protocolo de transferencia de hipertexto.
- **SSH**. Protocolo que permite gestionar remotamente a otra computadora de la red de forma segura.

- **NNTP.** Protocolo de Transferencia de Noticias (en inglés Network News Transfer Protocol).
- **IRC**. Chat Basado en Internet (en inglés Internet Relay Chat)
- **DNS**. Protocolo para traducir direcciones de red.

2.1.- Comunicación entre aplicaciones.

TCP/IP funciona sobre el concepto del modelo cliente/servidor, donde, como recordarás de unidades anteriores:

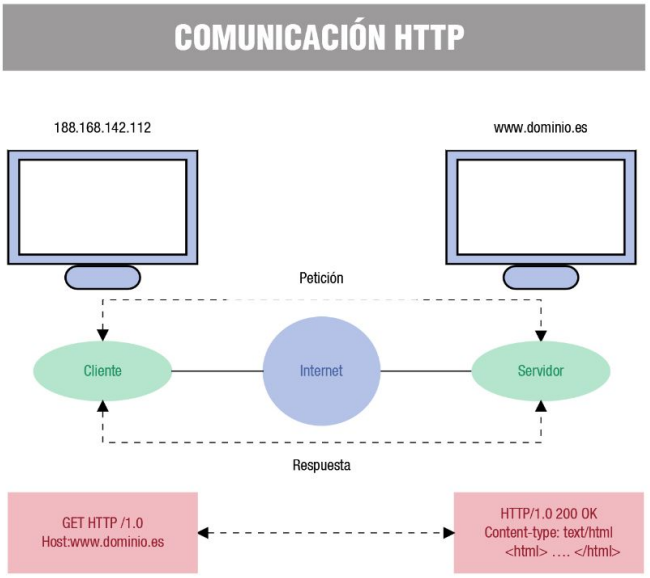
- **El Cliente** es el programa que ejecuta el usuario y solicita un servicio al servidor. Es quien inicia la comunicación.
- **El Servidor** es el programa que se ejecuta en una máquina (o varias) de red y ofrece un servicio (FTP, web, SMTP, etc.) a uno o múltiples clientes. Este proceso permanece a la espera de peticiones, las acepta y como respuesta, proporciona el servicio solicitado.

La capa de aplicación es el nivel que utilizan los programas para comunicarse a través de una red con otros programas. Los procesos que aparecen en esta capa, son aplicaciones específicas que pasan los datos al nivel de aplicación en el formato que internamente use el programa, y es codificado de acuerdo con un protocolo estándar. Una vez que los datos de la aplicación han sido codificados, en un protocolo estándar del nivel de aplicación, se pasan hacia abajo al siguiente nivel de la pila de protocolos TCP/IP.

En el nivel de transporte, las aplicaciones normalmente hacen uso de TCP y UDP, y son habitualmente asociados a un número de puerto. Por ejemplo, el servicio web mediante HTTP utiliza por defecto el puerto 80, el servicio FTP el puerto 21, etc.

Por ejemplo, la World Wide Web o Web, que usa el protocolo HTTP, sigue fielmente el modelo cliente/servidor:

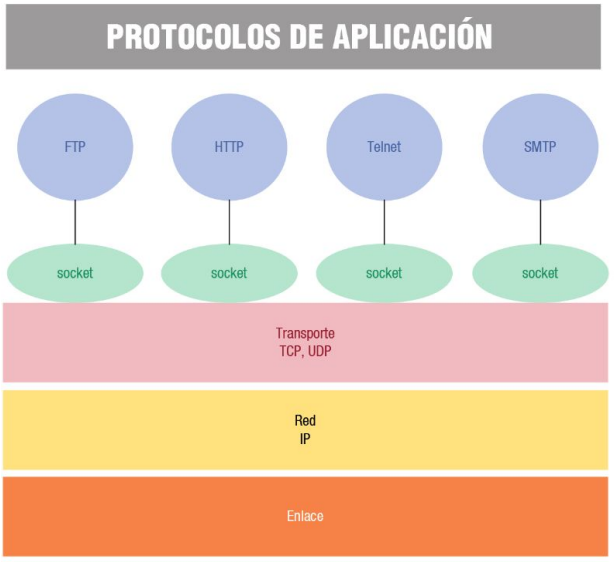
- Los clientes son las aplicaciones que permiten consultar las páginas web (navegadores) y los servidores, las aplicaciones que suministran (sirven) páginas web.
- Cuando un usuario introduce una dirección web en el navegador (una URL), por ejemplo **http://www.dominio.es**, éste solicita mediante el protocolo HTTP la página web al servidor web que se ejecuta en la máquina donde está la página.
- El servidor web envía la página por Internet al cliente (navegador) que la solicitó, y éste último la muestra al usuario.



Los dos extremos dialogan siguiendo un protocolo de aplicación, tal y como puedes ver en la imagen que representa el ejemplo anterior.

- El cliente solicita el recurso web **www.dominio.es** mediante **GET HTTP/1.0**
- El servidor le contesta **HTTP/1.0 200 OK** (todo correcto) y le envía la página html solicitada.

2.2.- Conexión, transmisión y desconexión.



Los protocolos de aplicación se comunican con el nivel de transporte mediante un API, denominada API Socket, y que en en el caso de Java viene implementada mediante las clases del paquete **java.net** como **Socket** y **ServerSocket**.

Como recordarás, un socket Java es la representación de una conexión para la transmisión de información entre dos ordenadores distintos o entre un ordenador y él mismo. Esta abstracción de medio nivel, permite despreocuparse de lo que está pasando en capas inferiores.

Dentro de una red, un socket es único pues viene caracterizado por cinco parámetros: el protocolo usado (HTTP, FTP, etc.), dos direcciones IP (la del equipo local o donde reside el proceso cliente, y la del equipo remoto o donde reside el proceso servidor) y dos puertos (uno local y otro remoto)

Te recordamos los pasos que se siguen para establecer, mantener y cerrar una conexión TCP/IP:

- Se crean los sockets en el cliente y el servidor
- El servidor establece el puerto por el que proporciona el servicio
- El servidor permanece a la escucha de las peticiones de los clientes.
- Un cliente conecta con el servidor.
- El servidor acepta la conexión.
- Se realiza el intercambio de datos.
- El cliente o el servidor, o ambos, cierran la conexión.

2.3.- DNS y resolución de nombres.

Todas las computadoras y dispositivos conectados a una red TCP/IP se identifican mediante una dirección IP, como por ejemplo **117.142.234.125.**

En su forma actual (IPv4), una dirección IP se compone de cuatro bytes sin signo (de 0 a 254) separados por puntos para que resulte más legible, tal y como has visto en el ejemplo anterior. Por supuesto, se trata de valores ideales para los ordenadores, pero para los seres humanos que quieran acordarse de la IP de un nodo en concreto de la red, son todo un problema de memoria.

El sistema DNS o Sistema de Nombres de Dominio es el mecanismo que se inventó, para que los nodos que ofrecen algún tipo de servicio interesante tengan un nombre fácil de recordar, lo que se denomina un nombre de dominio, como por ejemplo **www.todofp.es**.

DNS es un sistema de nomenclatura jerárquica para computadoras, servicios o cualquier recurso conectado a Internet o a una red privada.

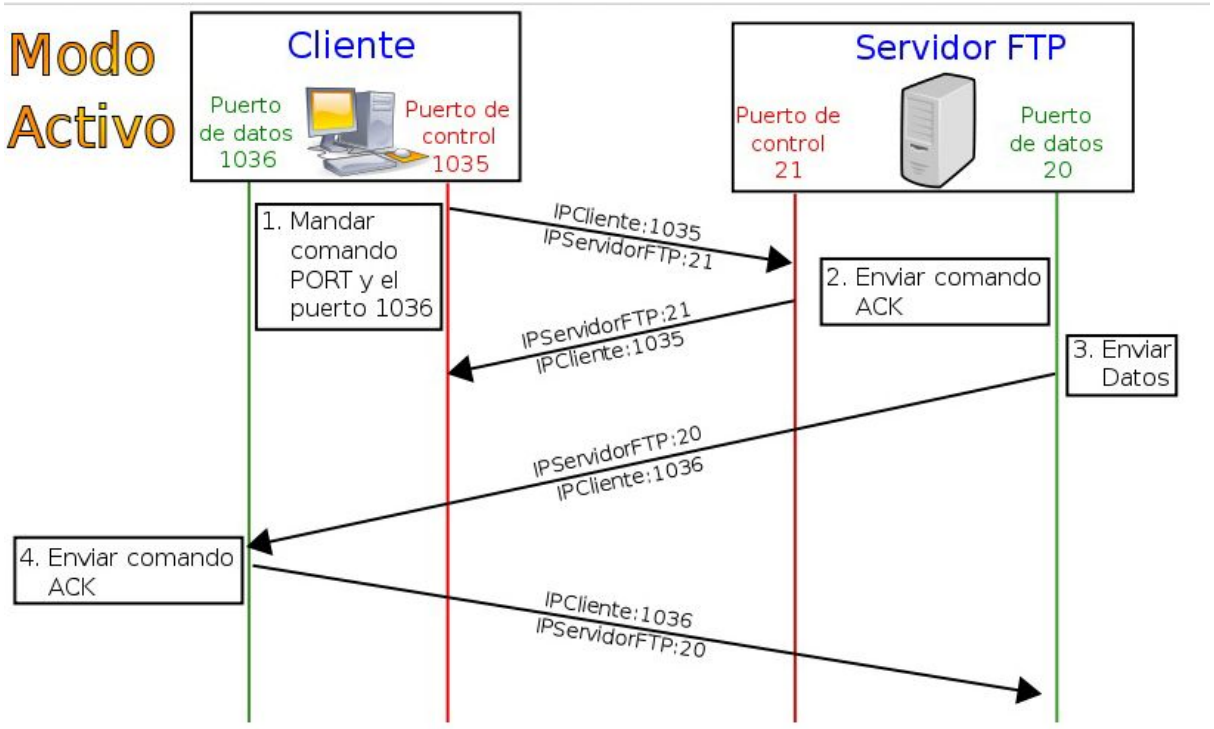
El objetivo principal de los nombres de dominio y del servicio DNS, es traducir o resolver nombres (por ejemplo **www.dominio.es**) en las direcciones IP (identificador binario) de cada equipo conectado a la red, con el propósito de poder localizar y direccionar estos equipos dentro de la red.

Sin la ayuda del servicio DNS, los usuarios de una red TCP/IP tendrían que acceder a cada servicio de la misma utilizando la dirección IP del nodo.

Además **el servicio DNS proporciona otras ventajas**:

- Permite que una misma dirección IP sea compartida por varios dominios.
- Permite que un mismo dominio se corresponda con diferentes direcciones IP.
- Permite que cualquier servicio de red pueda cambiar de nodo, y por tanto de IP, sin cambiar de nombre de dominio.

2.4.- El protocolo FTP.



El protocolo FTP o Protocolo de Transferencia de Archivos proporciona un mecanismo estándar de transferencia de archivos entre sistemas a través de redes TCP/IP.

Las **principales prestaciones de un servicio basado en el protocolo FTP o servicio FTP** son las siguientes:

- Permite el intercambio de archivos entre máquinas remotas a través de la red.
- Consigue una conexión y una transferencia de datos muy rápidas.

Sin embargo, **el servicio FTP adolece de una importante deficiencia en cuanto a seguridad**:

- La información y contraseñas se transmiten en texto plano. Esto está diseñado así, para conseguir una mayor velocidad de transferencia. Al realizar la transmisión en texto plano, un posible atacante puede capturar este tráfico, acceder al servidor y/o apropiarse de los archivos transferidos.

Este problema de seguridad, se puede solucionar mediante la encriptación de todo el tráfico de información, a través del protocolo no estándar SFTP usando SSH o del protocolo FTPS usando SSL. De encriptación ya hablaremos más adelante, en otra unidad de este módulo.

¿Cómo funciona el servicio FTP? Es una servicio basado en una arquitectura cliente/servidor y, como tal, seguirá las pautas generales de funcionamiento de este modelo, en ese caso:

- El servidor proporciona su servicio a través de dos puertos:
  - El puerto 20, para transferencia de datos.
  - El puerto 21, para transferencia de órdenes de control, como conexión y desconexión.
- El cliente se conecta al servidor haciendo uso de un puerto local mayor de 1024 y tomando como puerto destino del servidor el 21.

Las **principales características del servicio FTP** son las siguientes:

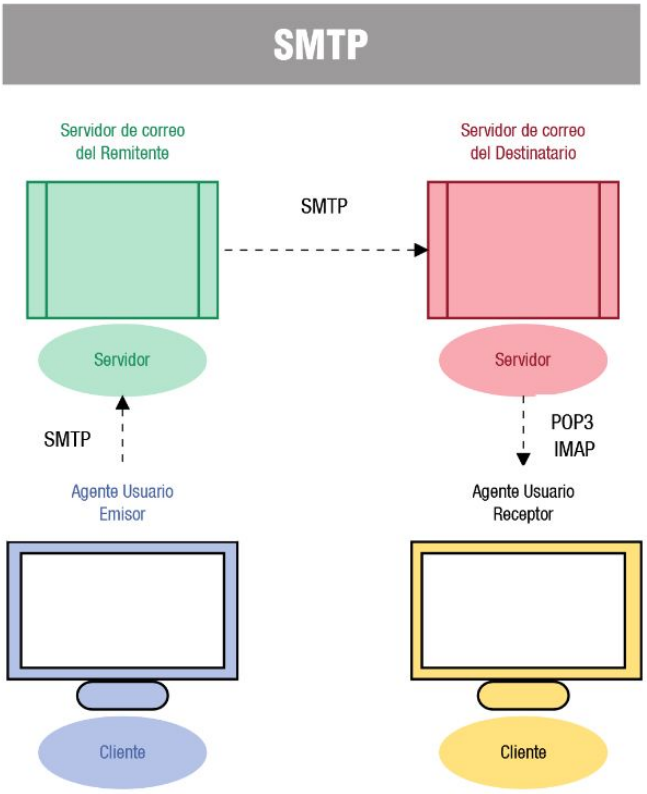
- La **conexión de un usuario remoto** al servidor FTP puede hacerse como: usuario del sistema (debe tener una cuenta de acceso), usuario genérico (utiliza la cuenta anonymous, esto es, usuario anónimo), usuario virtual (no requiere una cuenta local del sistema).
- El **acceso al sistema de archivos** es más o menos limitado, según el tipo de usuario que se conecte y sus privilegios. Por ejemplo, el usuario anónimo

sólo tendrá acceso al directorio público que establece el administrador por defecto.

- El servicio FTP soporta dos **modos de conexión**: modo activo y modo pasivo.
    - **Modo activo**. Es la forma nativa de funcionamiento (esquema ampliable de la derecha).
        - Se establecen **dos conexiones distintas**: la petición de transferencia por parte del cliente y la atención a dicha petición, iniciada por el servidor. De manera que, si el cliente está protegido con un cortafuegos, deberá configurarlo para que permita esta petición de conexión entrante a través de un puerto que, normalmente, está cerrado por motivos de seguridad.

    - **Modo pasivo**. El cliente sigue iniciando la conexión, pero el problema del cortafuegos se traslada al servidor.

2.5.- Los protocolos SMTP y POP3.



Esquema protocolos SMTP.

El **correo electrónico es un servicio que permite a los usuarios enviar y recibir mensajes y archivos rápidamente a través de la red**.

- Principalmente se usa este nombre para denominar al sistema que provee este servicio en Internet, mediante el protocolo SMTP o Protocolo Simple de Transferencia de Correo, aunque por extensión también puede verse aplicado a sistemas análogos que usen otras tecnologías.
- Por medio de mensajes de correo electrónico se puede enviar, no solamente texto, sino todo tipo de documentos digitales.

El servicio de correo basado en el protocolo SMTP sigue el modelo cliente/servidor, por lo que el trabajo se distribuye entre el programa Servidor y el Cliente. Te indicamos a continuación, **algunas consideraciones importantes sobre el servicio de correo a través de SMTP**:

- El servidor mantiene las cuentas de los usuarios así como los buzones correspondientes.
- Los clientes de correo gestionan la descarga de mensajes así como su elaboración.
- El servicio SMTP utiliza el puerto 25.
- El protocolo SMTP se encarga del transporte del correo saliente desde la máquina del usuario remitente hasta el servidor que almacene los mensajes de los destinatarios.
    - El usuario remitente redacta su mensaje y lo envía hacia su servidor de correo.
    - Desde allí se reenvía al servidor del destinatario, quién lo descarga del buzón en la máquina local mediante el protocolo POP3, o la consulta vía web, haciendo uso del protocolo IMAP.

2.6.- El protocolo HTTP.

El protocolo HTTP o Protocolo de Transferencia de Hipertexto es un conjunto de normas que posibilitan la comunicación entre servidor y cliente, permitiendo la transmisión de información entre ambos. La información transferida son las conocidas páginas HTML o páginas web. Se trata del método más común de intercambio de información en la World Wide Web.

HTTP define tanto la sintaxis como la semántica que utilizan clientes y servidores para comunicarse. Algunas **consideraciones importantes sobre HTTP** son las siguientes:

- Es un protocolo que sigue el **esquema petición-respuesta** entre un cliente y un servidor.
- Utiliza por defecto el **puerto 80**

- Al **cliente que efectúa la petición**, por ejemplo un navegador web, se le conoce como agente del usuario (user agent).
- A **la información transmitida se la llama recurso y se la identifica mediante un localizador uniforme de recursos (URL)**.
- **Los recursos pueden ser** archivos, el resultado de la ejecución de un programa, una consulta a una base de datos, la traducción automática de un documento, etc.

El **funcionamiento esquemático del protocolo HTTP** es el siguiente:

- El usuario especifica en el cliente web la dirección del recurso a localizar con el siguiente formato: **http://dirección[:puerto][ruta]**, por ejemplo: **http://www.iesalandalus.org/organizacion.htm**
- El cliente web descompone la información de la URL diferenciando el protocolo de acceso, la IP o nombre de dominio del servidor, el puerto y otros parámetros si los hubiera.
- El cliente web establece una conexión al servidor y solicita el recurso web mediante un mensaje al servidor, encabezado por un método, por ejemplo **GET /organizacion.htm HTTP/1.1** y otras líneas.
- El servidor contesta con un mensaje encabezado con la línea **HTTP/1.1 200 OK**, si existe la página y la envía, o bien envía un código de error.
- El cliente web interpreta el código HTML recibido.
- Se cierra la conexión.

HTTP es un protocolo sin estado, lo que significa que no recuerda nada relativo a conexiones anteriores a la actual. Algunas aplicaciones necesitan que se guarde el estado y para ello hacen uso de lo que se conoce como cookie.

3.- Bibliotecas de clases y componentes Java.

Java se ha construido con extensas capacidades de interconexión TCP/IP y soporta diferentes niveles de conectividad en red, facilitando la creación de aplicaciones cliente/servidor y generación de servicios en red. Así por ejemplo: permite abrir una URL como por ejemplo **http://www.dominio.es/public/archivo.pdf**, permite realizar una invocación de métodos remotos, RMI, o permite trabajar con sockets.

El paquete principal que proporciona el API de Java para programar aplicaciones con comunicaciones en red es:

- **java.net**. Este paquete soporta clases para generar diferentes servicios de red, servidores y clientes.

Otros paquetes Java para comunicaciones y servicios en red son::

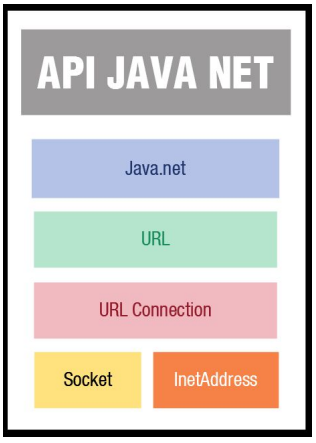
- **java.rmi**. Paquete que permite implementar una interface de control remoto (RMI).
- **javax.mail**. Permite implementar sistemas de correo electrónico.

Para ciertos servicios estándar, Java no proporciona objetos predefinidos y por tanto una solución fácil para generarlos es recurrir a bibliotecas externas, como por ejemplo, el API proporcionado por Apache Commons Net (las bibliotecas **org.apache.commons.net**). Esta API permite implementar aplicaciones cliente para los protocolos estándar más populares, como: Telnet, FTP, o FTP sobre HTTP entre otros.

3.1.- Objetos predefinidos.

El **paquete** **java.net**, proporciona una API que **se puede dividir en dos niveles**:

- **Una API de bajo nivel**, que permite representar los siguientes objetos:
  - **Direcciones**. Son los identificadores de red, esto es, las direcciones IP.
    - Clase **InetAddress**. Implementa una dirección IP.
  - **Sockets**. Son los mecanismos básicos de comunicación bidireccional de datos.
    - Clase **Socket**. Implementa un extremo de una conexión bidireccional.
    - Clase **ServerSocket**. Implementa un **socket** que los servidores pueden utilizar para escuchar y aceptar peticiones de clientes.
    - Clase **DatagramSocket**. Implementa un **socket** para el envío y recepción de datagramas.
    - Clase **MulticastSocket**. Representa un **socket** datagrama, útil para enviar paquetes multidifusión.
  - **Interfaces**. Describen las interfaces de red.



    - Clase **NetworkInterface**. Representa una interfaz de red compuesta por un nombre y una lista de direcciones IP asignadas a esta interfaz.
- **Una API de alto nivel**, que se ocupa de representar los siguientes objetos, mediante las clases que te indicamos:
    - **URI**. Representan los identificadores de recursos universales.
        - Clase **URI**.
    - **URL**. Representan localizadores de recursos universales.
        - Clase **URL**. Representa una dirección URL.
    - **Conexiones**. Representa las conexiones con el recurso apuntado por URL.
        - Clase **URLConnection**. Es la superclase de todas las clases que representan un enlace de comunicaciones entre la aplicación y una URL.
        - Clase **HttpURLConnection**. Representa una **URLConnection** con soporte para HTTP y con ciertas características especiales.

Mediante las clases de alto nivel se consigue una mayor abstracción, de manera que, esto facilita bastante la creación de programas que acceden a los recursos de la red. A continuación veremos algunos ejemplos.

3.2.- Métodos y ejemplos de uso de InetAddress.

La clase **InetAddress** proporciona objetos que puedes utilizar para manipular tanto direcciones IP como nombres de dominio. También proporciona métodos para resolver los nombres de host a sus direcciones IP y viceversa.

Una instancia de **InetAddress** consta de una dirección IP y en algunos casos también del nombre de host asociado. Esto último depende de si se ha creado con el nombre de host o bien ya se ha aplicado la resolución de nombres.

Esta clase no tiene constructores. Sin embargo, **InetAddress dispone de métodos estáticos que devuelven objetos InetAddress**. Te indicamos cuáles son esos métodos:

- **getLocalHost()**. Devuelve un objeto de tipo **InetAddress** con los datos de direccionamiento de mi equipo en la red local (no del conocido **localhost**).
- **getByName(String host)**. Devuelve un objeto de tipo **InetAddress** con los datos de direccionamiento del host que le pasamos como parámetro. Donde el parámetro **host** tiene que ser un nombre o IP válido, ya sea de Internet (como

iesalandalus.org o **195.78.228.161**), o de tu red de área local (como **documentos.servidor** o **192.168.0.5**). Incluso puedes poner **localhost**, o cualquier otra IP o nombre NetBIOS de tu red local.

- **getAllByName(String host).** Devuelve un **array** de objetos de tipo **InetAddress** con los datos de direccionamiento del host pasado como parámetro. Recuerda que en Internet es frecuente que un mismo dominio tenga a su disposición más de una IP.

Todos estos métodos pueden generar una excepción **UnknownHostException** si no pueden resolver el nombre pasado como parámetro.

Algunos **métodos interesantes de un objeto** **InetAddress** para resolver nombres son los siguientes:

- **getHostAddress()**. Devuelve en una cadena de texto la correspondiente IP.
- **getAddress()**. Devuelve un **array** formado por los grupos de bytes de la IP correspondiente.

Otros métodos interesantes de esta clase son:

- **getHostName()**. Devuelve en una cadena de texto el nombre del host.
- **isReachable(int tiempo)**. Devuelve TRUE o FALSE dependiendo de si la dirección es alcanzable en el tiempo indicado en el parámetro.

Tienes un ejemplo de uso de **InetAddress**. Con este ejemplo tratamos de ilustrar el uso de los métodos anteriores para resolver algunos nombres a su dirección o direcciones IP, tanto en la red local como en Internet. Por ello, para probarlo, debes tener conexión a Internet, en otro caso sólo se ejecutará la parte relativa a la red local, y se lanzará la correspondiente excepción al intentar resolver nombres de Internet.

3.3.- Programación con URL.

La programación de **URL** se produce a un nivel más alto que la programación de **sockets** y ésto, puede facilitar la creación de aplicaciones que acceden a recursos de la red.

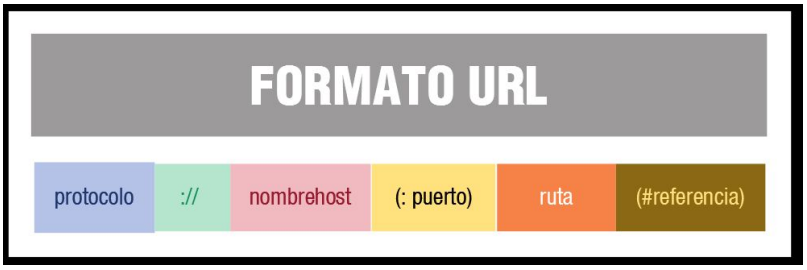
¿Tienes claro lo que es una URL y las partes en las que se puede descomponer? Vamos a verlo.

Una URL, Localizador Uniforme de Recursos, representa una dirección a un recurso de la World Wide Web. Un **recurso** puede ser algo tan simple como un archivo o un

directorio, o puede ser una referencia a un objeto más complicado, como una consulta a una base de datos, el resultado de la ejecución de un programa, etc.

Consideraciones sobre una URL:

- Puede referirse a sitios web, ficheros, sitios ftp, direcciones de correo electrónico, etc.
- La **estructura de una URL** se puede dividir en varias partes, tal y como puede ver en la imagen:



- **Protocolo**. El protocolo que se usa para comunicar.
- **Nombrehost**. Nombre del host que proporciona el servicio o servidor.
- **Puerto**. El puerto de red en el servidor para conectarse. Si no se especifica, se utiliza el puerto por defecto para el protocolo.
- **Ruta**. Es la ruta o path al recurso en el servidor.
- **Referencia**. Es un fragmento que indica una parte específica dentro del recurso especificado.

La clase **URL** de Java permite representar un URL. Utilizando la clase URL, se puede establecer una conexión con cualquier recurso que se encuentre en Internet, o en nuestra Intranet. Una vez establecida la conexión, invocando los métodos apropiados sobre ese objeto URL se puede obtener el contenido del recurso en el cliente. Esto es una idea potentísima, pues permite, en teoría, descargar el contenido de cualquier recurso en el cliente, incluso aunque se requiriera un protocolo que no existía cuando se escribió el programa.
Pero la clase URL también proporciona una forma alternativa de conectar un ordenador con otro y compartir datos, basándose en **streams**. Esto último, es algo que también se puede hacer con la programación de **sockets**, tal y como has visto anteriormente.

3.4.- Crear y analizar objetos URL.

¿Cómo podemos crear objetos URL? La clase **URL** dispone de diversos constructores para crear objetos tipo URL que se diferencian en la forma de pasarle la dirección URL. Por ejemplo, se puede crear un objeto **URL**:

- A partir de todos o algunos de los elementos que forman parte de una URL, como por ejemplo:
  URL url=new URL("http", "www.iesalandalus.org",80","index.htm"). Crea un objeto **URL** a partir de los componentes indicados (protocolo, host, puerto, fichero), esto es, crea la URL: **http://www.iesalandalus.org:80/index.htm**.
- A partir de la cadena especificada, dejando que el sistema utilice todos los valores por defecto que tiene definidos, como por ejemplo: **URL url=new URL("http://www.iesalandalus.org")** que crearía la URL : **http://www.iesalandalus.org**.
- A partir de una ruta relativa pasada como primer parámetro.
- A partir de las especificaciones indicadas y un manejador del protocolo que conoce cómo comunicarse con el servidor.

Cada uno de los constructores de URL puede lanzar una **MalformedURLException** si los argumentos del constructor son nulos o el protocolo es desconocido. Lo normal, es capturar y manejar esta excepción mediante un bloque **try- catch**.

Las URLs son objetos de una sola escritura. Lo que significa, que una vez que has creado un objeto URL no se puede cambiar ninguno de sus atributos (protocolo, nombre del host, nombre del fichero ni número de puerto).

Se puede analizar y descomponer una URL utilizando los siguientes métodos:

- getProtocol(). Obtiene el protocolo de la URL.
- getHost(). Obtiene el host de la URL.
- getPort(). Obtiene el puerto de la URL, si no se ha especificado obtiene -1.
- getDefaultPort(). Obtiene el puerto por defecto asociado a la URL, si no lo tiene obtiene -1.
- getFile(). Obtiene el fichero de la URL o una cadena vacía si no existe.
- getRef(). Obtiene la referencia de la URL o null si no la tiene.

3.5.- Leer y escribir a través de una URLConnection.

Un **objeto URLConnection se puede utilizar para leer desde y escribir hacia el recurso al que hace referencia el objeto URL.**

De entre los muchos **métodos que nos permiten trabajar con conexiones URL** vamos a centrarnos en primer lugar en los siguientes:

- **URL.openConnection()**. Devuelve un objeto **URLConnection** que representa una nueva conexión con el recurso remoto al que se refiere la URL.
- **URL.openStream()**. Abre una conexión a esta dirección URL y devuelve un InputStream para la lectura de esa conexión. Es una abreviatura de: **openConnection(). getInputStream ()**.

Te mostramos a continuación dos ejemplos muy sencillos que leen una URL, basándose tan solo en estos dos métodos. Los pasos a seguir para leer la URL son:

- Crear el objeto **URL** mediante **URL url=new URL(...)**;
- Obtener una conexión con el recurso especificado mediante **URL. openConnection()**.
- Abrir conexión con esa URL mediante **URL.openStream()**.
- Manejar los flujos necesarios para realizar la lectura.

También podemos utilizar objetos **URL** para leer una URL y analizar su código fuente, tal y como hacen los buscadores, optimizadores de código o validadores de código.

3.6.- Trabajar con el contenido de una URL.

El método que permite **obtener el contenido de un objeto URL** es:

- **URL.getContent()**. Devuelve el contenido de esa URL. Este método determina en primer lugar el tipo de contenido del objeto llamando al método **etContentType()**. Si esa es la primera vez que la aplicación ha visto ese tipo de contenido específico, se crea un manejador para dicho tipo de contenido.

Mediante este método y otros proporcionados por la clase **URLConnection**, veremos al final de este apartado, un ejemplo de cómo examinar el contenido del objeto URL y realizar la tarea que interese.

Crear una conexión URL realmente implica los siguientes pasos:

- Crear un objeto **URLConnection**, éste se crea cuando se llama al método **openConnection()** de un objeto **URL**.
- Establecer los parámetros y propiedades de la petición.
- Establecer la conexión utilizando el método **connect()**.
- Se puede obtener información sobre la cabecera o/y obtener el recurso remoto.

A partir de un objeto **URLConnection**, se puede obtener información muy útil sobre la conexión que se haya establecido y existen muchos métodos en la clase **URLConnection** que se pueden utilizar para examinar y manipular el objeto que se crea de este tipo.

Vamos a ver algunos métodos que serán de utilidad en nuestro ejemplo:

- **connect()**. Establece una conexión entre la aplicación (el cliente) y el recurso (el servidor), permite interactuar con el recurso y consultar los tipos de cabeceras y contenidos para determinar el tipo de recurso de que se trata.
- **getContentType()**. Devuelve el valor del campo de cabecera **content-type** o **null** si no está definido.
- **getContentLength()**. Devuelve el valor del campo de cabecera **content-length** o -1 si no está definido.
- **getLastModified()**. Devuelve la fecha de la última modificación del recurso.

4.- Programación de aplicaciones cliente.

La programación de aplicaciones cliente y aplicaciones servidor que vamos a realizar, está basada en el uso de los protocolos estándar de la capa de aplicación, por tanto, cuando programemos una aplicación servidor basada en el protocolo HTTP, por ejemplo, diremos también que se está programando un servicio o servidor HTTP, y si lo que programamos es el cliente, hablaremos de cliente HTTP.

Nos vamos a centrar en la programación de aplicaciones cliente para los protocolos HTTP, FTP, SMTP y TELNET. Su programación se realizará mediante bibliotecas especiales que proporcionan ciertos objetos que facilitan la tarea de programación, y en algunos casos veremos también cómo programar esas mismas aplicaciones cliente mediante **sockets**.

Es imprescindible un mínimo conocimiento de cómo funciona el protocolo sobre el que vamos a construir un cliente, para tener claro el intercambio de mensajes que

realiza con el servidor. Por tanto, no repares en volver a los primeros apartados o consultar los enlaces para saber más, donde se explica algo sobre el funcionamiento de estos protocolos. Esto es más necesario, si la programación se realiza a niveles relativamente bajos, como por ejemplo cuando utilizamos **sockets**.

4.1.- Programación de un cliente HTTP.

Como ya te hemos comentado anteriormente, HTTP se basa en sencillas operaciones de solicitud/respuesta.

- Un cliente establece una conexión con un servidor y envía un mensaje con los datos de la solicitud.
- El servidor responde con un mensaje similar, que contiene el estado de la operación y su posible resultado.

Al programar aplicaciones con las clases **URL** y **URLConnection**, lo hacemos en un nivel alto, de manera que queda oculta toda la operatoria que era tan explícita al programar un cliente con **sockets**.

4.2.- Bibliotecas para programar un cliente FTP.

Java no dispone de bibliotecas específicas para programar clientes y servidores de FTP. Pero afortunadamente, la organización de software libre The Apache software Foundation proporciona una API para implementar clientes FTP en tus aplicaciones.

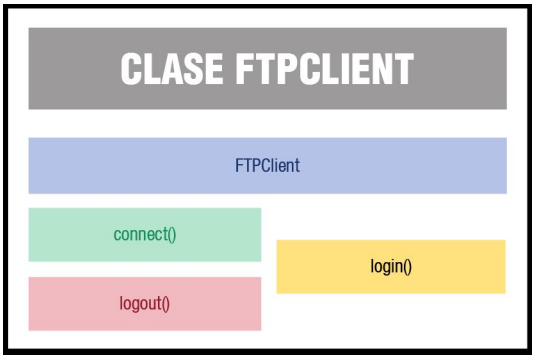
El **paquete de la API de Apache para trabajar con FTP es org.apache.commons.net.ftp.** Este paquete proporciona, entre otras, las siguientes **clases:**

- **Clase FTP**. Proporciona las funcionalidades básicas para implementar un cliente FTP. Esta clase hereda de **SocketClient**.
- **Clase FTPReplay**. Permite almacenar los valores devueltos por el servidor como código de respuesta a las peticiones del cliente.
- **Clase FTPClient**. Encapsula toda la funcionalidad que necesitamos para almacenar y recuperar archivos de un servidor FTP, encargándose de todos los detalles de bajo nivel para su interacción con el servidor. Esta clase hereda de **SocketClient**.
- **Clase FTPClientConfig**. Proporciona un forma alternativa de configurar objetos **FTPClient**.

- **Clase FTPSClient**. Proporciona FTP seguro, sobre SSL. Esta clase hereda de **FTPClient**.

4.3.- Programación de un cliente FTP.

Una forma sencilla de **crear un cliente FTP** es **mediante la clase FTPClient.**



Una vez creado el cliente, como en cualquier objeto **SocketClient** habrá que seguir el siguente **esquema básico de trabajo**:

- **Realizar la conexión del cliente en el servidor**. El método **connect(InetAddress host)** realiza la conexión del cliente con el servidor de nombre **host**, abriendo un objeto **Socket** conectado al host remoto en el puerto por defecto.
- **Comprobar la conexión**. El método **getReplyCode()** devuelve un código de respuesta del servidor indicativo de el éxito o fracaso de la conexión.
- **Validar usuario**. El método **login(String usuario, String password)** permite esa validación.
- **Realizar operaciones contra el servidor**. Por ejemplo:
  - Listar todos los ficheros disponibles en una determinada carpeta remota mediante el método **listNames()**.
  - Recuperar el contenido de un fichero remoto mediante **retrieveFile(String rutaRemota, OutputStream ficheroLocal)** y transferirlo al equipo local para escribirlo en el **ficheroLocal** especificado.
- **Desconexión del servidor**. El método **disconnect()** o **logout()** realiza la desconexión del servidor.

Ten en cuenta, que durante todo este proceso puede generarse tanto una **SocketException** si se superó el tiempo de espera para conectar con el servidor, o una **IOException** si no se tiene acceso al fichero especificado.

Tienes un ejemplo de programación de un cliente FTP, que se conecta a un servidor remoto y se descarga un fichero. Ten en cuenta que para poder ejecutarlo, puede que tengas que desactivar cualquier cortafuegos o firewall que tengas activo.

4.4.- Programación de un cliente Telnet.

Telnet (TELecommunication NETwork) es un **protocolo que permite acceder a otro equipo de la red y administrarlo de forma remota, esto es, como si estuviéramos sentados delante de él**.

El protocolo Telnet está basado en:

- El modelo cliente/servidor, por lo que su esquema básico de funcionamiento será el típico de esta arquitectura.
- El servidor escucha las peticiones por el puerto 23.
- Su funcionamiento es en modo texto.

Este servicio puede resultar muy útil para administrar por ejemplo equipos sin pantalla o teclado, o bien servidores apilados en un rack o que no estén físicamente presentes.

En la actualidad, no se suele utilizar, debido a la poca seguridad que ofrece, ya que toda la información que se intercambia entre servidor y cliente, incluidos usuario y contraseña, se transmiten en texto plano. Esto es así, porque Telnet se creó pensando en la facilidad de uso y no en la seguridad. En su lugar, se utiliza un servicio de acceso y control remoto basado en el protocolo SSH.

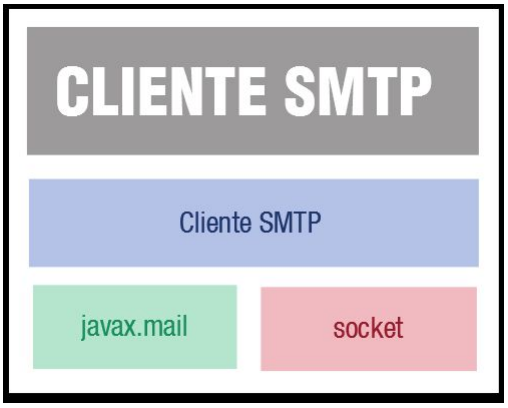
Veremos un ejemplo de **programación de un cliente Telnet, sólo a modo ilustrativo de uso de otra biblioteca proporcionada por el API de Apache,** **org.apache.commons.net**, que ya te descargaste en los apartados anteriores.

La biblioteca necesaria es **org.apache.commons.net.telnet**. Para disponer de ella, tendremos que agregar a las bibliotecas del proyecto, como en el ejemplo de FTP, el archivo **commons-net-n.x.x.jar**. Entre las clases que proporciona para programar un cliente Telnet está:

- **Clase** **TelnetClient.** Permite implementar un terminal virtual para el protocolo Telnet. Hereda de la clase **SocketClient**.
    - El método **SocketClient.connect()** realiza la conexión al servidor .
    - Los métodos **TelnetClient.getInputStream()** y **TelnetClient.getOutputStream()** permiten a través de objetos **InputStream()** y **OutputStream()** enviar y recibir datos a través de la conexión Telnet.

- El método **TelnetClient.disconnect()** cierra la conexión al servidor, así como los flujos de entrada y salida y el **socket** abiertos.

4.5.- Programación de un cliente SMTP.



Vamos a ver dos ejemplos de programación de clientes SMTP. Uno de ellos basado en **sockets** y otro realizado mediante el API **javax.mail**.

En el segundo caso, utilizaremos el API **javax.mail**. Este paquete proporciona las clases necesarias para implementar un sistema de correo. Vamos a destacar las **clases y métodos del paquete javax.mail que nos permitirán crear nuestro cliente de correo:**

- **Clase Session**. Representa una sesión de correo. Agrupa las propiedades y valores por defecto que utiliza el API **javax.mail** para el correo.
  - Método **getDefaultInstance()**. Obtiene la sesión por defecto. Si no ha sido configurada, se creará una nueva de manera predeterminada. El parámetro que se le pasa debe recoger al menos las siguientes propiedades: protocolo y servidor **smtp**, puerto para el **socket** de sesión y tipo, usuario y puerto **smtp**).
- **Clase Message**. Modela un mensaje de correo electrónico.
  - Método **setFrom()**. Asigna el atributo **From** al mensaje, siendo éste la dirección del emisor.
  - Método **setRecipients()**. Asigna el tipo y direcciones de destinatarios.
  - Método **setSubject()**. Para indicar asunto del mensaje.
  - Método **setText()**. Asigna el texto o cuerpo del mensaje.
- **Clase Transport.** Representa el transporte de mensajes. Hereda de la clase **Service**, la cual proporciona funcionalidades comunes a todos los servicios de mensajería, tales como conexión, desconexión, transporte y almacenamiento.
  - Método **send()**. Realiza el envío del mensaje a todas las direcciones indicadas. Si alguna dirección de destino no es válida, se lanza una excepción **SendFailedException**.

5.- Programación de servidores.

Entre los diferentes aspectos que se deben tener en cuenta cuando se diseña o se programa un servidor o servicio en red, vamos a resaltar los siguientes:

- El servidor debe poder **atender a multitud de peticiones que pueden ser concurrentes en el tiempo**. Esto lo podemos conseguir mediante la programación del servidor utilizando hilos o Threads.
- Es importante **optimizar el tiempo de respuesta del servidor**. Esto lo podemos controlar mediante la monitorización de los tiempos de proceso y transmisión del servidor.

La clase **ServerSocket** es la que se utiliza en Java a la hora de crear servidores. Para programar servidores o servicios basados en protocolos del nivel de aplicación, como por ejemplo el protocolo HTTP, será necesario conocer el comportamiento y funcionamiento del protocolo de aplicación en cuestión, y saber que tipo de mensajes intercambia con el cliente ante una solicitud o petición de datos.

En los siguientes apartados vamos a ver el ejemplo de cómo programar un servidor web básico, al que después le añadiremos la funcionalidad de que pueda atender de manera concurrente a varios usuarios, optimizando los recursos.

5.1.- Programación de un servidor HTTP.

Antes de lanzarnos a la programación del servidor HTTP, vamos a recordar o conocer cómo funciona este protocolo y a sentar las hipótesis de trabajo.

El servidor que vamos a programar cumple lo siguiente:

- Se basa en la versión 1.1 del protocolo HTTP.
- Implementará solo una parte del protocolo.
- Se basa en dos tipos de mensajes: peticiones de clientes a servidores y respuestas de servidores a clientes.
- Nuestro servidor solo implementará peticiones **GET**.

Para crear un servidor HTTP o servidor web, el esquema básico a seguir será:

- Crear un socketServer asociado al puerto 80 (puerto por defecto para el protocolo HTTP).

- Esperar peticiones del cliente.
- Acceptar la petición del cliente.
- Procesar petición (intercambio de mensajes según protocolo + transmisión de datos).
- Cerrar socket del cliente.

A continuación, vamos a programar un sencillo servidor web que acepte peticiones por el puerto 8066 de un cliente que será tu propio navegador web. Según la URL que incluyas en el navegador, el servidor contestará con diferente información. Los casos que vamos a contemplar son los siguientes:

- Al poner en tu navegador http://locahost:8066, te dará la bienvenida.
- Al poner en tu navegador http://localhost:8066/quijote, mostrará un párrafo de el Quijote.
- Al poner en tu navegador una URL diferente a las anteriores, como por ejemplo http://localhost:8066/a, mostrará un mensaje de error.

Recuerda detener o parar el servidor, una vez lo hayas probado, antes de volver reiniciarlo.

5.2.- Implementar comunicaciones simultáneas.

Para proporcionar la funcionalidad a nuestro servidor, de que pueda atender comunicaciones simultáneas es necesario utilizar hilos o threads.

Un servidor HTTP realista tendrá que atender varias peticiones simultáneamente. Para ello, tenemos que ser capaces de modificar su código para que pueda utilizar varios hilos de ejecución. Esto se hace de la siguiente manera:

- El hilo principal (o sea, el que inicia la aplicación) creará el **socket** servidor que permanecerá a la espera de que llegue alguna petición.
- Cuando se reciba una, la aceptará y le asignará un **socket** cliente para enviarle la respuesta. Pero en lugar de atenderla él mismo, el hilo principal creará un nuevo hilo para que la despache por el **socket** cliente que le asignó. De esta forma, podrá seguir a la espera de nuevas peticiones.

Esquemáticamente, **el código del hilo principal tendrá el siguiente aspecto**:

```
try {
 socServidor = new ServerSocket(puerto);
   while (true) {
     //acepta una petición, y le asigna un socket cliente para la respuesta
     socketCliente = socServidor.accept();
     //crea un nuevo hilo para despacharla por el socketCliente que le asignó
     hilo = new HiloDespachador(socketCliente);
     hilo.start();
   }
 } catch (IOException ex) {
}
```

donde la clase **HiloDespachador** será una extensión de la clase **Thread** de Java, cuyo constructor almacenará el **socketCliente** que recibe en una variable local utilizada luego por su método **run()** para tramitar la respuesta:

```
class HiloDespachador extends Thread {
   private socketCliente;
   public HiloDespachador(Socket socketCliente) {
   this.socketCliente = socketCliente;
 }
 public void run() {
```

```
  try{
    //tramita la petición por el socketCliente
  } catch (IOException ex) {
  }
 }
}
```

Hecho esto, tendremos todo un Servidor HTTP capaz de gestionar peticiones concurrentes de manera más eficiente.

5.3.- Monitorización de tiempos de respuesta.

Una cuestión muy importante para evaluar el rendimiento y comportamiento de una aplicación cliente/servidor es monitorizar los tiempos de respuesta del servidor. Los tiempos que intervienen desde que un cliente realiza una petición al servidor hasta que recibe su resultado son dos:

- **Tiempo de procesamiento**. Es el tiempo que el servidor necesita para procesar la petición del cliente y enviar los datos.
- **Tiempo de transmisión**. Es el tiempo que tardan los mensajes en llegar a su destino a través de la red.

¿Cómo podemos medir esos tiempos?

Para **medir el tiempo de procesamiento** bastar medir el tiempo que transcurre en el servidor para procesar la solicitud del cliente. Por ejemplo, el siguiente código permite medir ese tiempo en milisegundos:

```java
 //anota la lectura del reloj del Sistema antes de inciar el procesamiento
 long tiempoInicio = System.currentTimeMillis();
//procesamiento de la petición del cliente

//anota la lectura del reloj del Sistema al finalizar el procesamiento
 long tiempoFin= System.currentTimeMillis();
 long tiempoProceso= tiempoFin - tiempoInicio;
System.out.println("Tiempo en procesar petición: "+ tiempoProceso);
```

Para **medir el tiempo de transmisión** será necesario que el servidor envíe un mensaje con el tiempo del sistema al cliente. El cliente al recibir el mensaje debe calcular su tiempo de sistema y compararlo con el tiempo recibido en el mensaje.

Para poder comparar los tiempos de respuesta de dos equipos es imprescindible que sus relojes de sistema estén sincronizados a través de cualquier servicio de tiempo (NTP). En equipos Windows por ejemplo, la sincronización de los relojes se realiza automáticamente.